DOCUMENT RESUME

ED 057 579 EM 009 419

AUTHOR	Feurzeig, Wallace; And Others
TITLE	An Introductory LOGO Teaching Sequence: LOGO Teaching Sequence on Logic, LOGO Reference Manual.
INSTITUTION	Bolt, Beranek and Newman, Inc., Cambridge, Mass.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	R-2165
PUB DATE	30 Jun 71
NOTE	135p.; Programming-Languages as a Conceptual Framework for Teaching Mathematics, Volume One; See also EM 009 420, EM 009 421, EM 009 422
EDRS PRICE	MF-$0.65 HC-$6.58
DESCRIPTORS	*Computer Assisted Instruction; Geometry; Logic; Manuals; *Mathematics Instruction; *Programing Languages
IDENTIFIERS	Project LOGO

ABSTRACT

In earlier work a programing language, LOGO, was developed to teach mathematics in the framework of computer programs. Using LOGO a few programs were tested in both elementary and junior high school mathematics classrooms with excellent results. The work reported here is the first effort to systematically develop extensive curriculum materials using the LOGO language. This first volume gives a reference manual on the LOGO language and two of the LOGO teaching sequences. The short introductory sequence, on teletype geometry, is one of many possible starting points for studying LOGO. The sequence on logic is the most advanced of those presented; it has a sophisticated ability for dealing with syllogistic-type arguments. For subsequent volumes see EM 009 420, EM 009 421, and EM 009 422. (JY)

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

Report No. 2165 Volume 1

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

An Introductory LOGO Teaching Sequence

LOGO Teaching Sequence on Logic

LOGO Reference Manual

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, N.W.
Washington, D. C. 20550

Contract NSF-C 615

30 June 1971

CAMBRIDGE NEW YORK CHICAGO LOS ANGELES

PROGRAMMING-LANGUAGES AS A CONCEPTUAL FRAMEWORK FOR TEACHING MATHEMATICS

Final Report on the second fifteen months of the LOGO Project

Wallace Feurzeig
George Lukas
Philip Faflick
Richard Grant
Joan D. Lukas
Charles R. Morgan
Walter B. Weiner
Paul M. Wexelblat

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Massachusetts 02138

30 June 1971

Submitted to:

National Science Foundation
Office of Computing Activities
1800 G Street, NW
Washington, D. C. 20550

Contract NSF-C 615

CONTENTS

PREFACE

This is the final report of work under Contract NSF-C 615, "Programming-Languages as a Conceptual Framework for Teaching Mathematics." In earlier directly-related work, supported by the Office of Computing Activities, we advanced the thesis that mathematics could be developed and presented in the framework of programs, and that this kind of presentation would greatly enhance teaching and learning. Using the LOGO programming language designed expressly for this purpose, we tested this thesis in both elementary and junior high school mathematics classrooms, obtaining the excellent results we had anticipated.*

The teaching materials used in this earlier work were developed "on the run" to meet immediate class needs. These materials were, therefore, unconnected or loosely connected segments of partially realized designs. The focus of this phase of the work was on classroom experimentation and testing, not on curriculum development. It was clear to us from the outset, however, that extensive curriculum material would eventually be needed. The object of the work reported here is to take the first steps in the development of such materials.

For this purpose, we have developed five extended LOGO teaching sequences and an introductory LOGO sequence. These range over a variety of elementary mathematical subjects, levels of difficulty, and mathematical and pedagogic style. This diversity is intentional. We want to illustrate the scope of this new approach to mathematics and its general accessibility to teachers and students.

---

*"Programming-Languages as a Conceptual Framework for Teaching Mathematics," Final Report on the first fifteen months of the LOGO Project, Wallace Feurzeig et al, Nov. 1969.

The report is composed of four volumes. Volume 1 contains a reference manual on the LOGO language and two of the LOGO teaching sequences. The short introductory sequence, on teletype geometry, is one of many good starting points for studying LOGO. The LOGO sequence on logic is by far the most advanced of those presented here. In it we develop a sophisticated ability for dealing with syllogistic-type arguments. The sequence demonstrates that an extremely complex set of procedures can be evolved in a natural way.

Volume 2 includes two LOGO teaching sequences treating some standard mathematical topics -- on the representation of numbers and the algorithms of arithmetic, and on functions and equations -- but incorporating the new viewpoints made possible by the use of LOGO programs. The sequence on numbers is the most detailed presentation of the series in accordance with its mathematical content. Their content and level of difficulty makes these two sequences well-suited for incorporation into existing curricula.

Volume 3 is comprised of two LOGO teaching sequences on "problem-solving". One deals with the generation and testing of methods and strategies. The other treats the problem of translating between different representations, both formal and informal. The first sequence uses a variety of mathematical contexts; the second uses the context of story problems in algebra.

We do not regard the teaching sequences as literal teaching prescriptions. Rather, we designed them as exemplary materials to acquaint prospective teachers with the rich mathematical and pedagogical possibilities inherent in the use of LOGO. The sequences are intended as source materials for assisting teachers in the preparation of courses. We think the presentations will

be useful in providing teachers general guidelines around which to make their own variations and extensions.

Volume 4 of this report stands apart from the other three. It contains a detailed description of the LOGO processor algorithm. It was written for system programmers and others interested in the details of LOGO's inner workings. It will facilitate the implementation of LOGO on computer systems of many kinds.

The main authors of the material in this report are Wallace Feurzeig, George Lukas, and Richard Grant. Philip Faflick made major contributions to the number and strategy sequences. Joan Lukas is a co-author of the logic sequence. Charles R. Morgan contributed both to the curriculum material and the LOGO system programming and documentation. Primary work in the latter area was done by Walter B. Weiner with the assistance of Paul M. Wexelblat. The demanding technical typing and drawing tasks were directed and performed by Pearl Stockwell.

Volume 1, Part 1

AN INTRODUCTORY LOGO TEACHING SEQUENCE

The LOGO Project

NSF-C 615

Wallace Feurzeig
George Lukas

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

AN INTRODUCTORY LOGO TEACHING SEQUENCE

We introduce students to LOGO by developing some procedures for making pictures, specifically "geometric" figures, on a teletypewriter. To begin with, we write procedures like the following, for drawing a rectangle:

```
TO RECTANGLE
1 PRINT "++++"
2 PRINT "++++"
3 PRINT "++++"
END
```

To use this procedure, we simply type

RECTANGLE

(When we use a procedure we will underscore our typing to distinguish it from the computer's response)

and LOGO responds:

```
++++
++++
++++
```

But this kind of procedure is pointillistic -- the figure is being specified point by point. To improve upon this, we can write procedures for drawing line segments of specified length to serve as building blocks for drawing a number of different kinds of figures. For example, the following procedures:

```
TO 1+
1 PRINT "+"
END
```

```
TO 2+
1 PRINT "++"
END
```

```
TO 3+
1 PRINT "+++"
END
```

```
TO 4+
1 PRINT "++++"
END
```

type segments of a single + mark, 2 + marks, 3 + marks, and 4 + marks, respectively.

These procedures are immediately used as parts of other, more interesting procedures for drawing figures of fixed shape and size. For example, we can describe a two-by-four rectangle,

```
TO RECTANGLE
1 4+
2 4+
END
```

To draw the rectangle, we type:

```
RECTANGLE
++++
++++
```

To make a larger rectangle we write:

```
TO BIG-RECTANGLE
1 4+                          (draw a row)
2 BIG-RECTANGLE               (and repeat the process)
END
```

This procedure will go on indefinitely:

BIG-RECTANGLE

```
++++
++++
++++
++++
++++
++++
++++
...
...
...
```

until we forcibly stop it, using the teletype "break" key.

To make a rectangle of specified size, say /N/ rows, we rewrite BIG-RECTANGLE as follows:

```
TO BIG-RECTANGLE /N/
1 TEST IS /N/ Ø                    (If all rows are done)
2 IF TRUE STOP                     (Stop)
3 4+                               (Otherwise, draw a row)
4 BIG-RECTANGLE (DIFF OF /N/ AND 1)  (and repeat the process
END                                    /N/-1 more times)
```

This allows us to draw rectangles with any desired number of rows, for example:

BIG-RECTANGLE 5

```
++++
++++
++++
++++
++++
```

We can also write procedures for drawing triangles:

```
TO TRIANGLE
1 1+
2 2+
3 3+
4 4+
END
```

TRIANGLE

```
+
++
+++
++++
```

But we run into difficulty when we try to extend this to make arbitrarily large triangles. We can accomplish this, however, by writing a single procedure MARK /N/ to type out any specified number /N/ of + marks. By varying the input of this new procedure, we can draw triangles as well as many other regular figures.

In fact, we will write two drawing procedures, BLANK /N/ and MARK /N/. The first of these types out /N/ blank spaces; the second types /N/ +'s and returns the carriage. These two procedures are virtually identical in form. Thus, for example:

```
TO MARK /N/                              (If there are no more marks to typ
1 TEST IS /N/ Ø
2 IF TRUE SKIP                           (skip to the next line)
3 IF TRUE STOP                           (and stop)
4 TYPE "+"                               (Otherwise, type a mark)
5 MARK (DIFFERENCE OF /N/ AND 1)(and repeat the process /N/-1
END                                         more times)
```

MARK 17

```
+++++++++++++++++
```

To illustrate the use of these basic procedures, let's write a procedure to make a little triangle, indented from the left margin.

```
TO MAKE-A-TRIANGLE
1 BLANK 1Ø
2 MARK 1
3 BLANK 9
4 MARK 3
5 BLANK 8
6 MARK 5
END
```

MAKE-A-TRIANGLE

```
    +
   +++
  +++++
```

This would be a tedious way of describing larger figures. A considerable improvement comes about from noting that in many cases we wish to center all rows of marks with respect to the same interval. We can easily write a procedure to do this. It will type /N/ marks centered in an interval of length /L/. The number of spaces it needs to indent before typing is given by the quotient of (/L/ - /N/) and 2. (We must compute this to the nearest integer since we cannot half-space on the teletype.) In LOGO this is expressed QUOTIENT OF (DIFFERENCE OF /L/ AND /N/) AND 2.

Now we can write the procedure MIDDLE /N/, for typing +'s in the middle /N/ spaces of an interval of length /L/.

```
TO MIDDLE /N/
1 BLANK QUOTIENT OF (DIFFERENCE OF /L/ AND /N/) AND 2
2 MARK /N/
END
```

Using MIDDLE we can now write MAKE-A-TRIANGLE without worrying about spacing.

```
TO MAKE-A-TRIANGLE
1 MIDDLE 1
2 MIDDLE 3
3 MIDDLE 5
END
```

MIDDLE has immediate and broad application as a basic procedure for drawing symmetric figures. To show its utility, let's first define a procedure for typing a rectangle with /A/ columns and /B/ rows.

```
TO RECTANGLE /A/ /B/
1 TEST IS /B/ Ø                                    (If /B/ has become 0,
2 IF TRUE STOP                                     stop)
3 MIDDLE /A/                                       (Else type /A/ +'s)
4 RECTANGLE /A/ (DIFFERENCE OF /B/ AND 1)          (and repeat the proces
END                                                /B/-1 times)
```

This procedure executes the command MIDDLE /A/ (which types /A/ "centered" +'s) /B/ times.

RECTANGLE 18 3

```
++++++++++++++++++
++++++++++++++++++
++++++++++++++++++
```

Rectangles are useful as basic building blocks for composing other figures. Trapezoids are also useful, particularly for building many kinds of polygons. And, they include triangles as a limiting case.

```
TO TRAPEZOID /A/ /B/    (/A/ and /B/ are the lengths of the two b
1 MIDDLE /A/                    (Type /A/ centered +'s)
2 TEST IS /A/ /B/               (If /A/ has become equal to /B/,
3 IF TRUE STOP                  stop)
4 TRAPEZOID (SUM OF /A/ AND 2) /B/   (Otherwise, repeat the proce
END                                  with /A/ increased by 2)
```

(Note that this procedure closely parallels that for RECTANGLE.)

TRAPEZOID 3 9

```
   +++
  +++++
 +++++++
+++++++++
```

TRIANGLE is the limiting case of TRAPEZOID, with /A/ = 1.

```
TO TRIANGLE /N/
1 TRAPEZOID 1 /N/
END
```

TRIANGLE 7

```
    +
   +++
  +++++
 +++++++
```

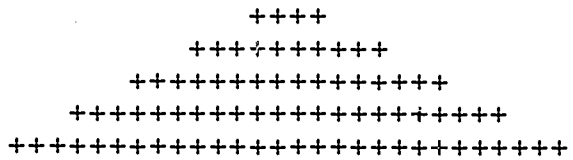
Though the TRAPEZOID procedure can be used to draw trapezoids of different sizes, it can only draw trapezoids with the same interior angles, because successive rows increase in width from a smaller to a larger base by a fixed step. TRAPEZOID can be generalized in a very straightforward way to incorporate a larger class of trapezoids. All we need do is include the step-size increment /STEP/ as an input. Doing this, the definition becomes,

```
TO TRAPEZOID /A/ /B/ /STEP/
1 MIDDLE /A/
2 TEST IS /A/ /B/
3 IF TRUE STOP
4 TRAPEZOID (SUM OF /A/ AND /STEP/) /B/ /STEP/
END
```

(The only changes have been the inclusion of /STEP/ in the title line and in line 4.)

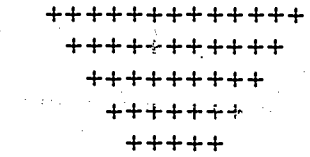
Now we can draw trapezoids with relatively big slopes:

TRAPEZOID 4 28 6



We can also draw "upside-down" trapezoids by using negative increments.

TRAPEZOID 13 5 -2



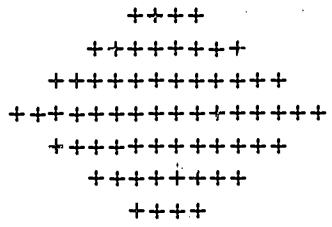
It is possible to specify values of /A/, /B/, and /STEP/ that do not properly define a trapezoid. For example, if the top base is to be 8 and the bottom base is to be 3, with step size +1, the drawing process will never terminate by itself. The procedure TRAPEZOID can easily be modified to check for this and all other nonterminating cases. Discovering and fixing such difficulties provides good problems for the students as a natural side-effect of their own work.

The power of the procedures we have just defined is evident from the ease with which we can use them to draw a large variety of other figures. Thus, a HEXAGON can be built from two trapezoids; a DIAMOND from two triangles; a PENTAGON from a trapezoid and a triangle; an OCTAGON from a trapezoid, a rectangle, and another trapezoid; and so on. For example, this is a procedure for drawing hexagons, where /A/ is the starting width and /B/ is the mid-width.

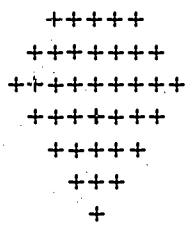
```
TO HEXAGON /A/ /B/ /STEP/
1 TRAPEZOID /A/ /B/ /STEP/
2 TRAPEZOID (DIFFERENCE OF /B/ AND /STEP/) /A/ (-/STEP/)
END
```

Here are examples of figures made by such procedures.

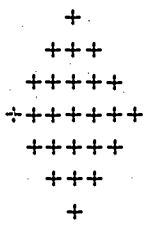
HEXAGON 4 16 4



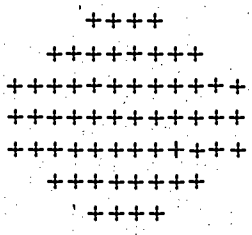
PENTAGON 5 9 2



DIAMOND 7 2

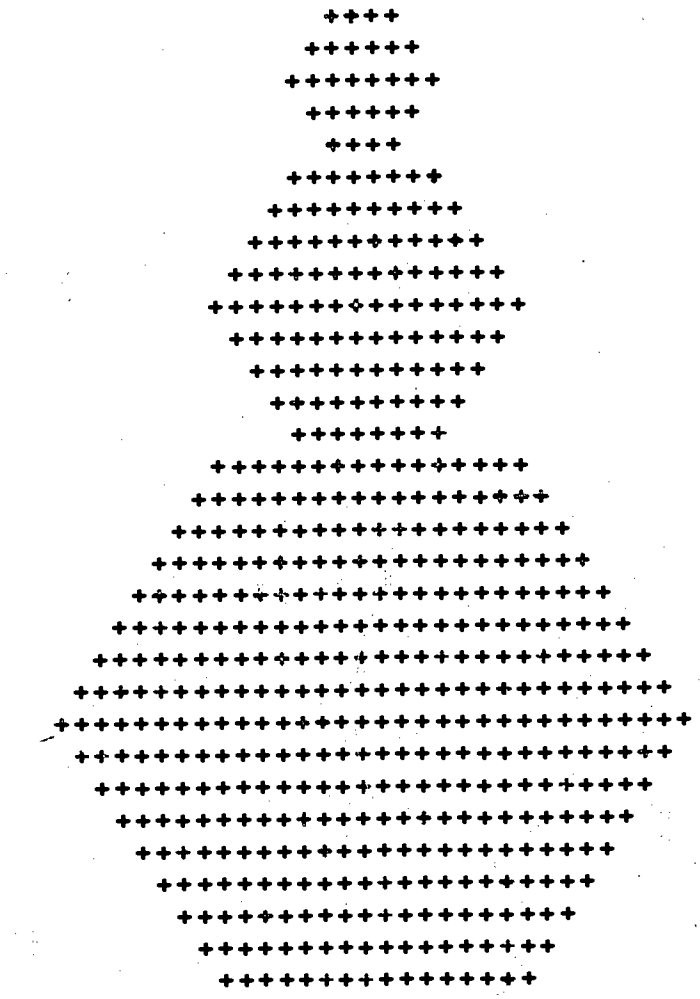


OCTAGON 4 12 4



These figure-drawing procedures can themselves be used as construction elements. For example, we can stack figures together to form towers such as the following one formed by hexagons of increasing size.

HEX-TOWER 4 8 3



The LOGO procedure for generating this kind of tower follows. (Note how similar it is in form to many previous ones.)

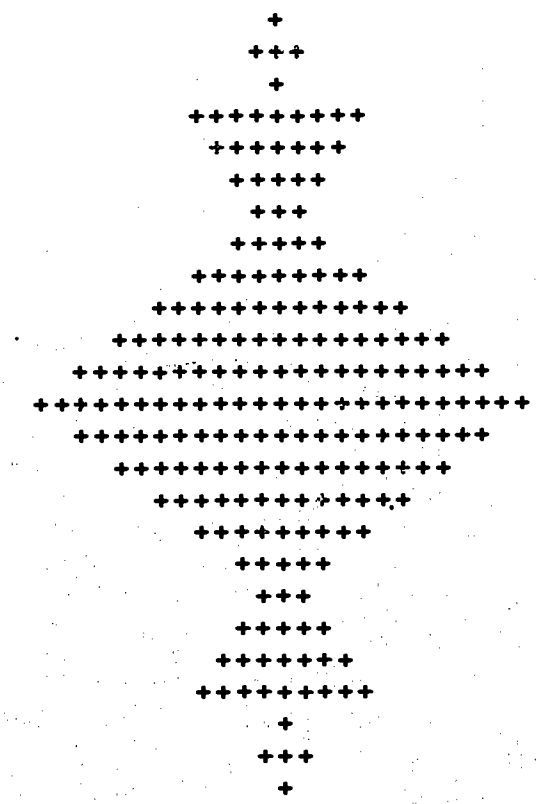
```
TO HEX-TOWER /A/ /B/ /NUM/          (/A/ and /B/ define the topmost
                                     hexagon, /NUM/ is the number of
                                     hexagons in the tower)

1 TEST IS /NUM/ Ø
2 IF TRUE STOP
3 HEXAGON /A/ /B/ 2                 (Draw a hexagon of dimensions
                                     /A/, /B/, 2)
4 HEX-TOWER (PRODUCT OF 2 AND /A/)  (Increase the hexagon size and
  (PRODUCT OF 2 AND /B/)             repeat the process (/NUM/ - 1)
  (DIFFERENCE OF /NUM/ AND 1)        times)
END
```

An interesting way of generating more complex symmetric structures is by using a random process to determine the constituent figures and their sequencing. We can develop a LOGO program, PATTERN /NAME/ /NUMBER/, to do this. PATTERN creates a drawing program, whose name is /NAME/, which uses /NUMBER/ figures to create a pattern that is symmetric about the horizontal as well as the vertical. Here, for example, are two especially nice LOGO procedures created by PATTERN, along with their resulting drawings.

PATTERN "TOTEM" 5

```
TO TOTEM
1 DIAMOND 3 2
2 TRAPEZOID 9 3 -2
3 HEXAGON 5 25 4
4 TRAPEZOID 3 9 2
5 DIAMOND 3 2
END
```

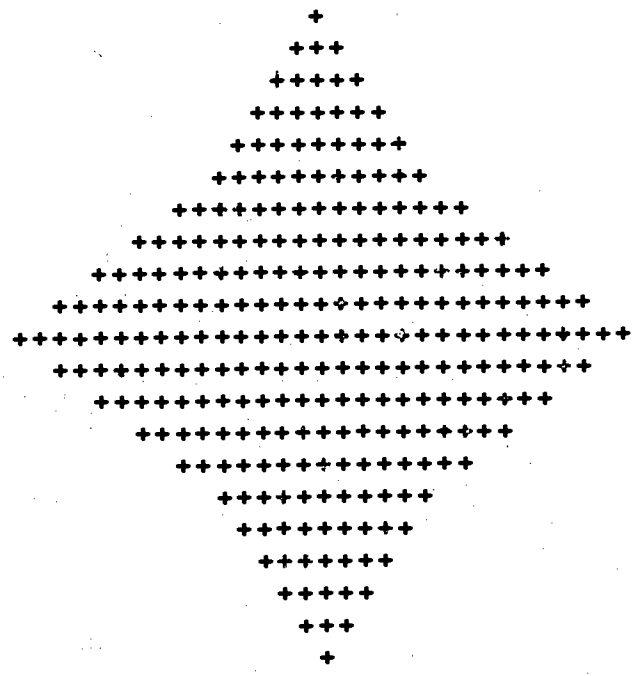
TOTEM



PATTERN "FIGURE-12" 3

```
TO FIGURE-12
1 TRIANGLE 11 2
2 HEXAGON 15 31 4
3 TRIANGLE 11 -2
END
```

FIGURE-12



Using the procedures written thus far, the student can generate a large number of different geometric shapes, and the writing of such a sequence of procedures represents a considerable achievement on his part. The foundations of geometry, however, lie in the transformation of geometric objects, not merely in their portrayal. Our next goal, then, is to write procedures for performing standard transformations of geometric figures. Such transformations include translation, rotation, and reflection, both with respect to a given point and with respect to a given

line. Procedures to generate the union and intersection of the sets of points defining two geometric objects are also useful. And we need a procedure DRAW which plots any given set of points.

To write these procedures we need a different representation for geometric objects, one which can be retained within the computer. (Clearly we do not have such a representation thus far -- our current objects are generated and drawn one line at a time.) Perhaps the simplest such representation is a list of pairs of numbers, each pair representing one point of the object. Then it is easy to write procedures, such as the following, which reflect a set of points about the x-axis.

```
TO REFLECTX /PAIR LIST/                (/PAIR LIST/ is the list of X Y
                                        number pairs)
1 TEST IS /PAIR LIST/ /EMPTY/          (Are there any points left on
2 IF TRUE OUTPUT /EMPTY/                /PAIR LIST/?  If not, terminate
                                        procedure)
3 OUTPUT LIST OF
    FIRST OF /PAIR LIST/
    NEGATIVE OF SECOND OF /PAIR LIST/
    REFLECTX OF (BUTFIRST2 OF /PAIR LIST/)
                                       (Otherwise, output a list of the
                                        X coordinate and the negative of
                                        the Y coordinate of the first
                                        number pair on /PAIR LIST/, and
                                        REFLECTX applied to the pair list
                                        obtained by deleting the first
END                                     number pair on /PAIR LIST/)
```

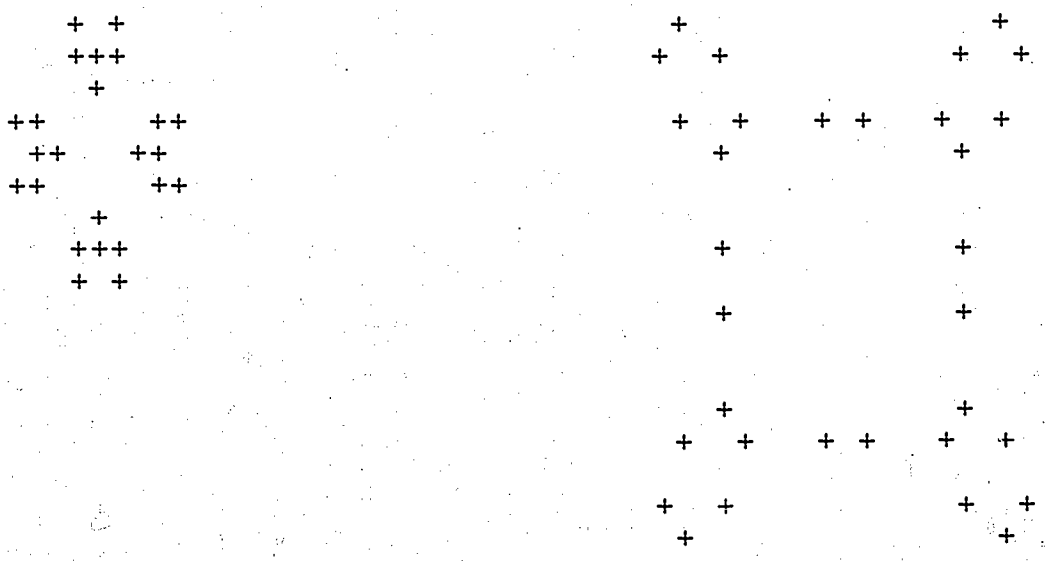
Thus for example:

```
PRINT REFLECTX OF "1 2 4 3 7 11 2 -1"
1 -2 4 -3 7 -11 2 1
```

Using this and two similar procedures, one for reflecting about the 45 degree line through the origin, and the other for reflecting about the Y-axis, we can now write a procedure for random generation of figures having eightfold symmetry.

```
TO EIGHTFOLD /N/                    (/N/ is the number of pairs on the
                                    pair list that will be generated)
1 MAKE "PAIR LIST" RANDOMLIST OF /N/
                                    (Generate a random list of /N/
                                    pairs, /PAIR LIST/)
2 MAKE "PAIR LIST"
   UNION OF (/PAIR LIST/) AND (REFLECT45 OF /PAIR LIST/)
                                    (Form the union of /PAIR LIST/ and
                                    the pair list formed by reflecting
                                    it around the 45 degree line, and
                                    make this the new /PAIR LIST/)
3 MAKE "PAIR LIST"
   UNION OF (/PAIR LIST/) AND (REFLECTY OF /PAIR LIST/)
                                    (Form the union of the new list and
                                    its reflection about the Y-axis)
4 MAKE "PAIR LIST"
   UNION OF (/PAIR LIST/) AND (REFLECTX OF /PAIR LIST/)
                                    (Do the same with the resulting list
                                    and its reflection about the x-axis)
5 DRAW ORDER OF /PAIR LIST/ "+"
                                    (Put the resulting pair list in
                                    lexicographic order and plot it
                                    using +'s)
END
```

EIGHTFOLD generates random drawings such as the following.



Using such basic transformation procedures we can study general Euclidean transformations. For example, we can develop methods for determining whether two randomly oriented polygons are congruent. Proceeding in this fashion we steadily progress from pre-mathematics to material of genuine mathematical content.

Volume 1, Part 2

LOGO TEACHING SEQUENCE ON LOGIC

Teacher's Text

and

Problems

The LOGO Project

NSF-C 615

Joan D. Lukas

George Lukas

Bolt Beranek and Newman Inc.
50 Moulton Street
Cambridge, Mass. 02138

CONTENTS

LOGO UNIT ON LOGIC

0. Introduction

Syllogisms formed the central core of logic until they were pushed aside by the development of mathematical logic in the Nineteenth Century. We have chosen to focus on them here because they are accessible without a great deal of formalism and because syllogistic types of argument occur frequently in everyday discourse.

This section develops a set of LOGO procedures which test the validity of syllogistic arguments. LOGO is peculiarly suited for such an application because of its non-numerical capabilities and its procedure-oriented programming heuristic. Algorithms in everyday language are first developed for each part of the rather extensive syllogism tester; the translation of these into LOGO procedures follows in a very natural manner. The method of testing syllogisms used is an adaptation of the one presented by Lewis Carroll in his book "Symbolic Logic".

The ideas underlying this section and the related background in symbolic logic were provided by Joan Lukas. George Lukas implemented this treatment in the form of LOGO procedures. Both shared in the actual writing.

1. Formal Background

Many of the inferences made in both formal and informal reasoning concern the relations among classes of objects. We may say, for instance, that *computers* can't *think* since *computers* are *machines* and *machines* can't *think*. While one may dispute one or the other of the premises, the structure of this argument is unassailable:

(1) All A are B
(2) All B are C
(3) ∴ All A are C

Once one has accepted (1) and (2), (3) inevitably follows.

If we had made a different statement about the classes, say that they overlap rather than that one is contained in the other, the argument is changed radically. We may not conclude that some babies are fifty years old if we know that some babies are male and some males are fifty years old. Here the structure of the argument is

(4) Some A are B
(5) Some B are C
(6) ∴ Some A are C

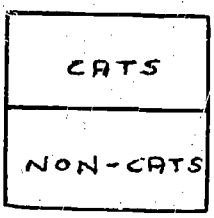
In this case (6) does not follow from (4) and (5).

Statements which express relations between classes, such as "All computers are machines," and "Some babies are male," are known as categorical statements. Arguments such as (1) (2) - (3) and (4) (5) - (6), whose premises and conclusions are categorical statements, are (categorical) syllogisms. The study of such statements and arguments and in particular the singling out of valid forms of syllogisms is a central concern of classical logic.

Categorical statements were first studied systematically by Aristotle who recognized four types of such statements. One of these asserts the containment of one class in another -- "All cats are animals" or "It always rains on Tuesday" (which can be translated as "Every Tuesday is a day when it rains"). By negating such a statement, we obtain a second type "Not all cats are animals" or "It doesn't always rain on Tuesday".

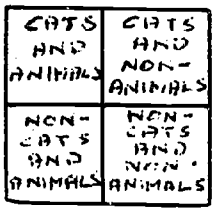
A third type of statement asserts that two classes have nothing in common: "No cats are dogs" "It never snows in August." Note that this statement is not equivalent to the statement above, i.e., "It never rains on Tuesday" is a stronger assertion than "It doesn't always rain on Tuesday." The fourth type of statement asserts that two classes do have something in common: "Some cats like milk" (or "*Some cats* are *things that like milk*").

The meaning of these types of statements, and the relations among them, can be made clearer by a diagrammatic representation. Consider the statement "Some cats are animals". We can imagine the universe partitioned into two classes -- cats and noncats.

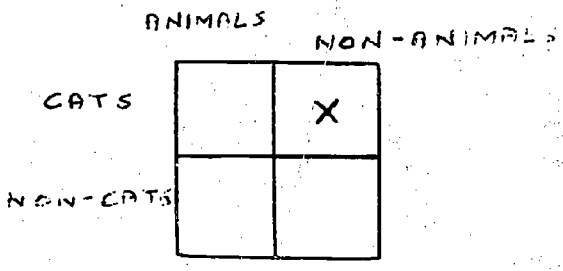


The universe can also be divided into animals and nonanimals.

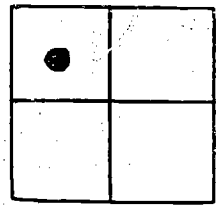
Lewis Carroll devised a means of representing the four possible classes obtained by combining these two partitions



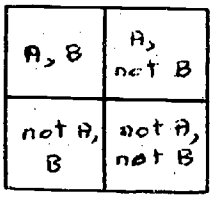
and using markers to indicate the existence or nonexistence of members of each class, as follows. An x in a region indicates that the condition represented by that region is never satisfied, while a • indicates that there are some objects (at least one object) which fall into the class. Thus, in the diagram above, an x in the upper right region indicates



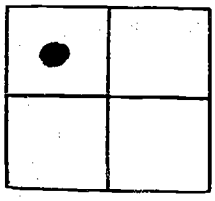
that there are no cats which are not animals and a • in the upper left region indicates that there are cats which are animals.



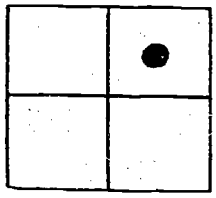
Similar diagrams can be used for any two classes A and B. If the right-hand side represents A, the left not A, the upper-half B, and the lower not B, we get



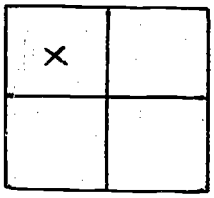
Then, "Some A are B" can be represented by



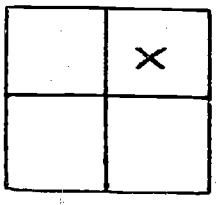
"Some A are not B" by



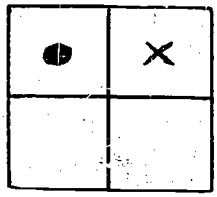
"No A are B" is represented by



In drawing the diagram corresponding to the statement "All A are B", we have to make a decision. It is clear that, if all A are B then no A are not B. So the diagram should have an x in the upper right region.



Is this the complete diagram for the statement, or does it also imply that there are some A's which are B's, so that a ● would appear in the upper left region?

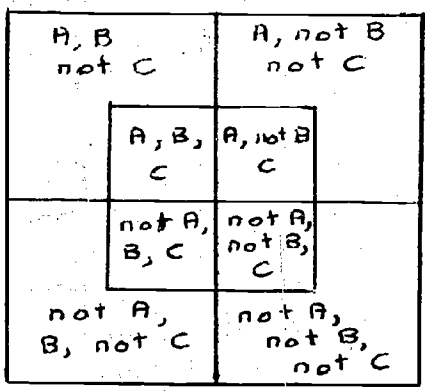


The circle seems to be appropriate because, if all A's are B's, than certainly some A's are B's. A difficulty arises, however, when we consider the possibility that there might not be any A's at all. Then "All A are B" might be taken as vacuously true. But, "Some A are B" would be false, since it implies the existence of A's.

Consider, for example, the statement "All round squares are pink." In most circumstances we would dismiss this statement as nonsense on the ground that there are no round squares and so there is no point in saying anything about them. If we were forced to assign a truth value to the statement, we might interpret it to mean both that round squares exist and every one of them is pink. Under this interpretation, the statement is false. Alternatively, we might rule it to be vacuously true, since it asserts that the class of round squares is contained in the class of pink objects and since the class of round squares is empty, it is contained in every class.

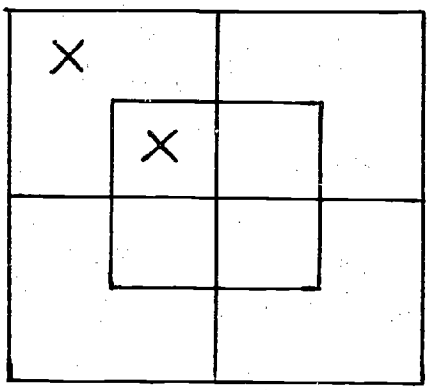
In mathematics a statement of the form "All A are B" is usually not taken to imply the existence of A's. We can talk meaningfully about the class of counter examples to Fermat's last theorem* without knowing whether any such counter examples exist. In ordinary language, however, this view can lead to absurdity. For instance, we would not want to accept the statement "All pink elephants are two feet tall" as true merely on the ground that there are no pink elephants. So we will take the diagram for "All A are B" to contain both an x in the A,not B region and a • in the A,B region, that is, the statement is taken as implying both "Some A are B" and "No A are not B".

In the study of categorical syllogisms, we are concerned with three classes of objects rather than two. The Lewis Carroll diagram is extended to three classes by dividing it into an inner square and an outer region, the inner square representing the third class, C, and the outer region the complement of this class, not C.

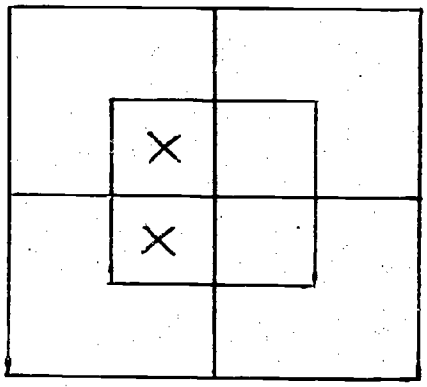


*Fermat's last theorem asserts that there are no integers x, y, z, and n with n>2 satisfying the equation $x^n+y^n=z^n$.

The diagram corresponding to the statement "No A are B" now becomes

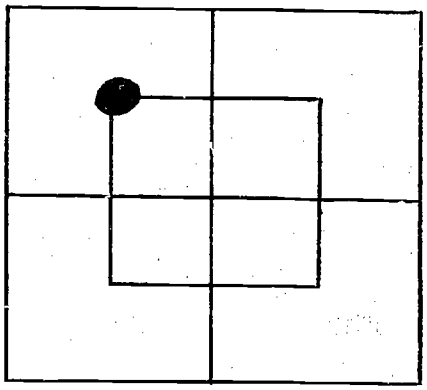


and "No B are C" has the diagram

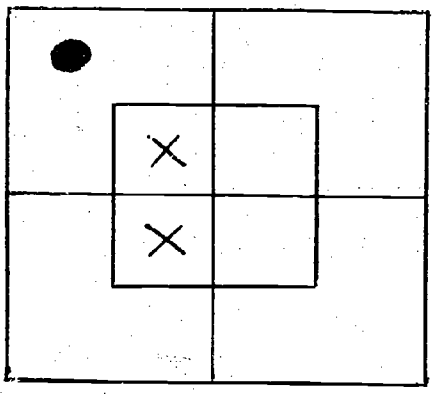


Clearly, two regions are excluded in each case.

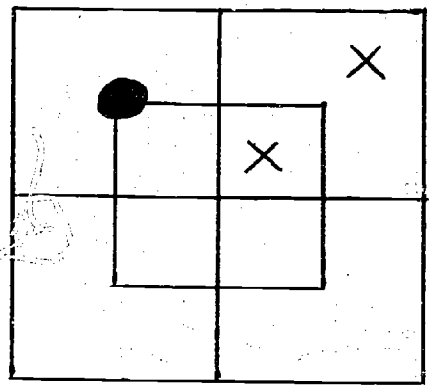
The statement "Some A are B" is now more difficult to diagram. It does not tell us whether the A-B objects belong to C or to its complement. We indicate this uncertainty by placing the circle on the line dividing the two regions A,B,C, and A,B,not C. It is "sitting on the fence".



Further statements may tell us which region the circle belongs in. For instance, "No B are C" would eliminate the inner region and force the circle into the upper left corner. The diagram resulting from these two statements is



We have interpreted "All A are B" to mean both "No A are not B" and "Some A are B" so the corresponding diagram is



We can now use these diagrams to check the validity of syllogistic arguments. A syllogism is valid if the premises imply the conclusion. In terms of diagrams, this means the diagram obtained by combining the premises contains x's and •'s wherever the conclusion's diagram does. The first syllogism we discussed was

(1) All computers are machines
(2) Machines can't think
(3) ∴ Computers can't think

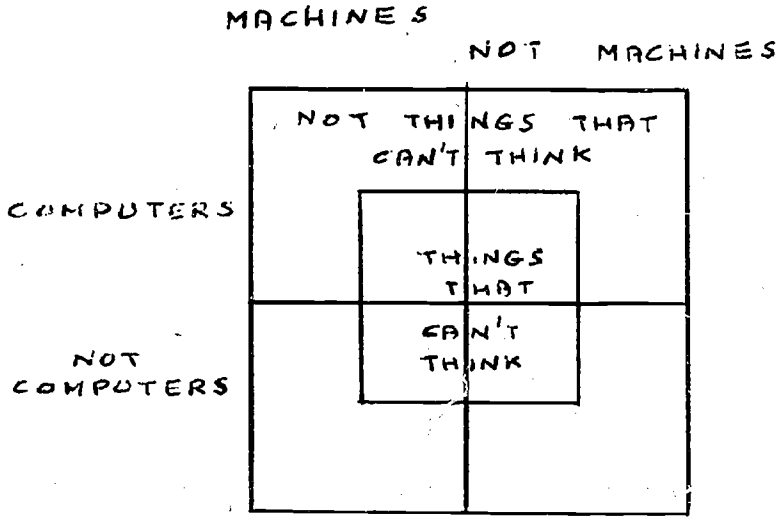
which was formalized as

(1) All A are B
(2) All B are C
(3) ∴ All A are C

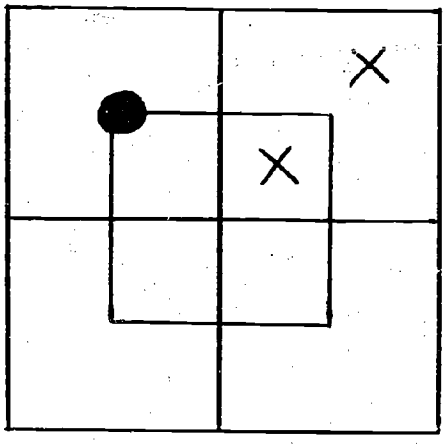
by setting A = the class of computers
B = the class of machines
C = the class of things that can't think.

Note that we have made explicit the "all" which was implicit in the statements and rewritten the second as "All machines are things that can't think". This is the usual procedure in treating syllogisms formally. If the verb in a statement is different

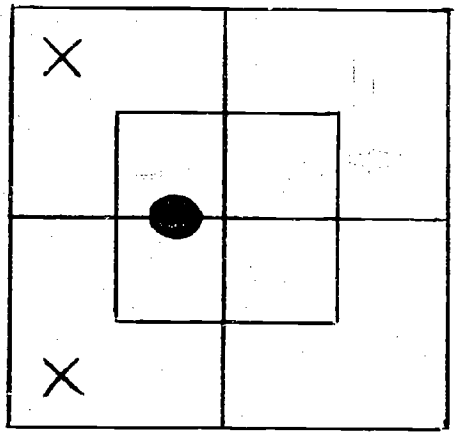
from "are" or "is", we translate the predicate into the form "are things that ____", e.g., "Birds fly" becomes "Birds are things that fly". We treat all statements in syllogisms as being about classes rather than about actions. If we form the diagram



(1) becomes

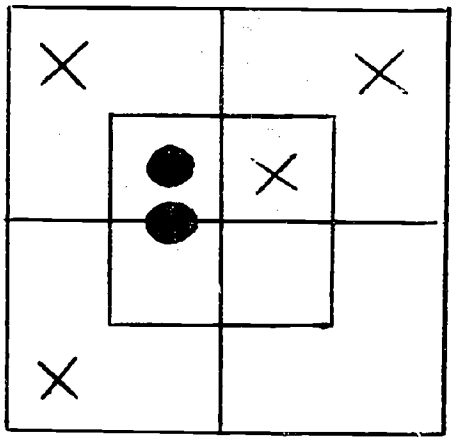


and (2) is



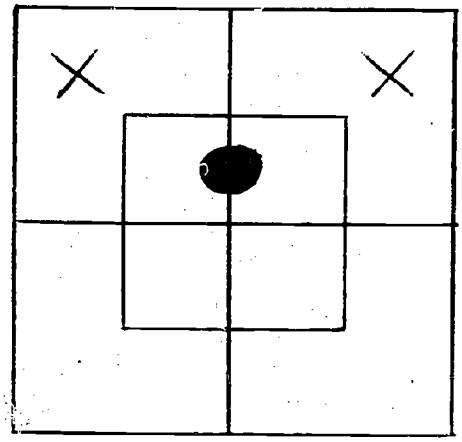
When these two diagrams are combined the x in the upper left corner forces the circle into the center square.

The combined diagram becomes



(The lower circle does not add any information to the diagram)

The conclusion of the argument is "All computers are things that can't think" which has the diagram



For this to be implied by the premises, the premises must give an x in the indicated places and a • in one of the two regions which intersect the circle in the diagram (or a circle intersecting both). As the combined diagram does satisfy these requirements, this syllogism is valid.

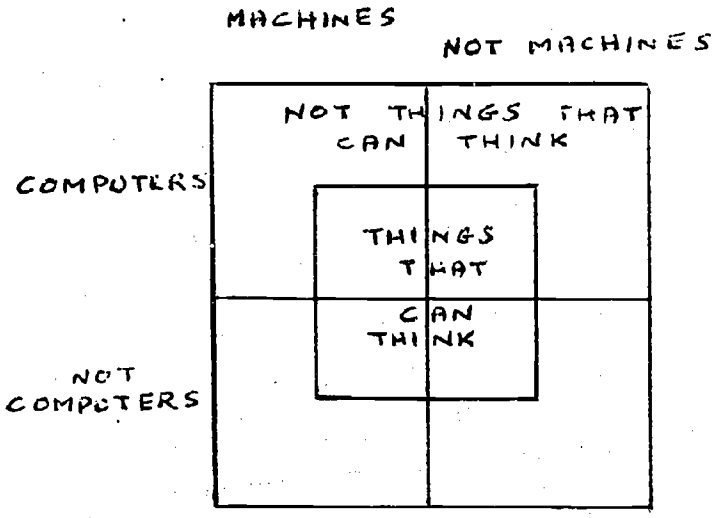
The same argument might be cast more naturally in a different syllogistic form if we make A = computers, B = machines, as before, but C = things that *can* think. Now statements 2 and 3 are replaced by (2') No machines are things that can think

(3') No computers are things that can think.

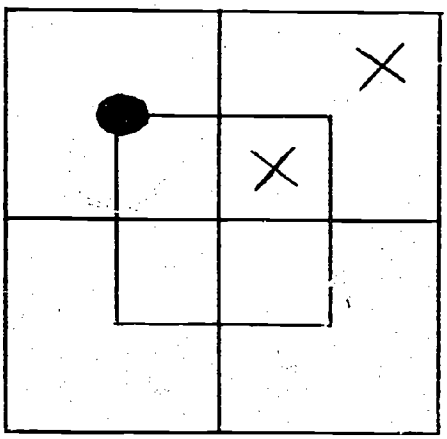
The form of the argument now becomes

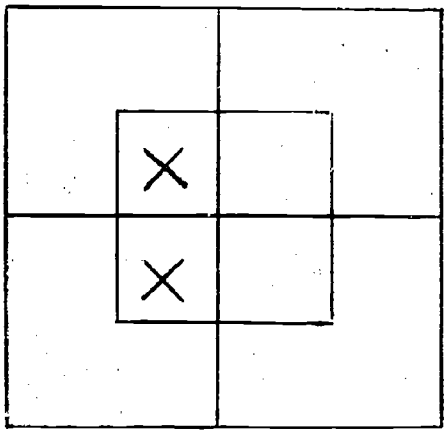
All A are B
No B are C
$\therefore$ No A are C

If we now construct the diagram as follows:

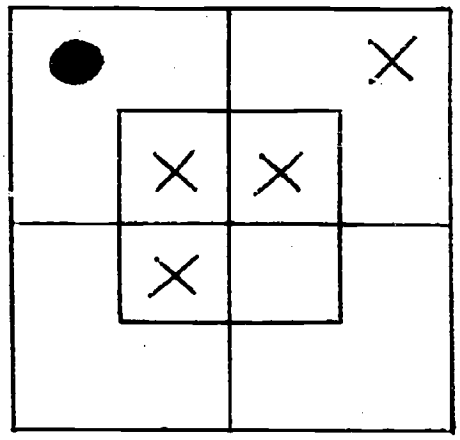


(1) is represented by and (2') by

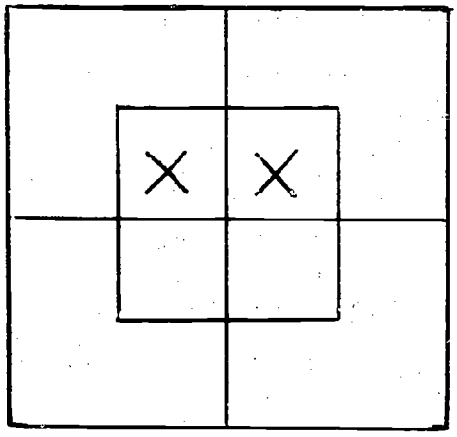




The combined diagram is



The diagram of (3') is



The syllogism is valid since the combined diagram does contain x's in the two indicated places.

Our second syllogism was

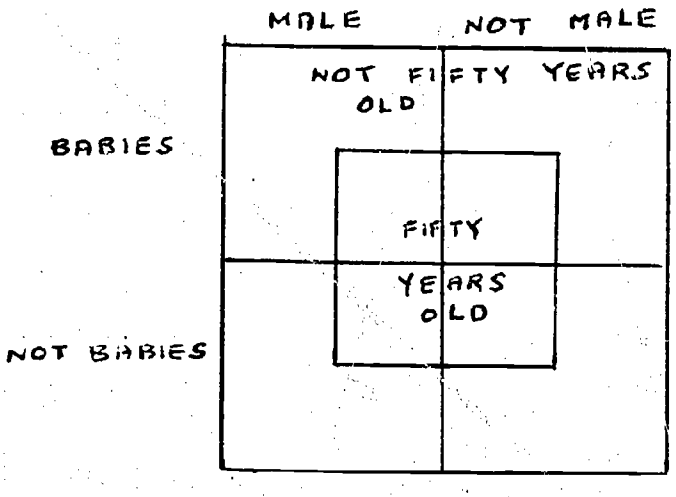
(4) Some babies are male
(5) Some males are fifty years old
(6) ∴ Some babies are fifty years old

This has the form

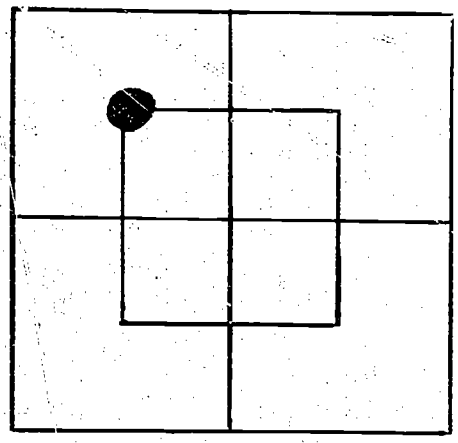
Some A are B
Some B are C
∴ Some A are C

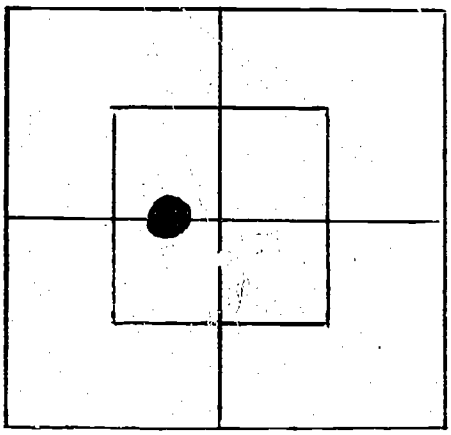
where A is the class of babies, B the class of males, and C the class of fifty-year-old people.

The diagram is

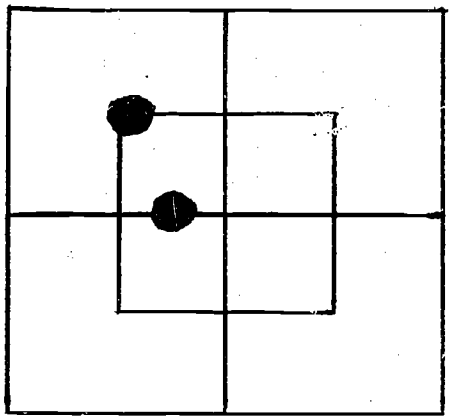


(4) is represented by and (5) by

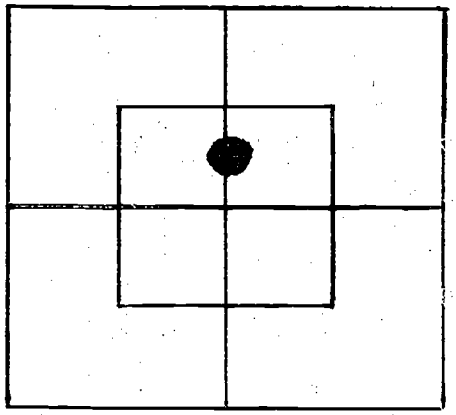




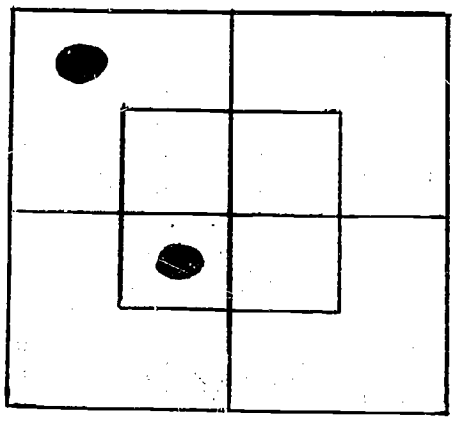
The combined diagram is



The diagram for (6) is



For the argument to be valid, the combined diagram must contain a circle in one of the two areas which intersect the circle in (6) or a circle intersecting both. This is not the case. In fact, the combined diagram is consistent with



which does not satisfy the conclusion.

Some inferences which contain statements about particular objects rather than classes can be treated as categorical syllogisms. For example,

John is a boy
All boys like ice cream
∴ John likes ice cream

can be formalized as

All A are B
All B are C
∴ All A are C

if we take A to be the class consisting of the single object "John", B the class of boys, and C the class of people who like ice cream.

An argument with a particular premise which is invalid is

John likes ice cream
Some people who like ice cream are fat
∴ John is fat

taking A = "John", as before,
B = people who like ice cream,
C = fat people.

This argument can be put in the form

All A are B
Some B are C
∴ All A are C

(Here we are using our knowledge that "John" usually refers to a person rather than a dog or some other thing.)

This syllogism can be seen to be invalid by constructing the Lewis Carroll diagrams for these three statements.

A syllogism is valid or invalid because of its form, not because of the truth or falsity of its premises and conclusion. For instance, the syllogism

All cows are things that fly
All things that fly are birds
∴ All cows are birds

is of the form All A are B
All B are C
All A are C

and hence valid, although its premises and conclusion are all false.

On the other hand, Some people are men
Some men have beards
∴ Some people have beards

is of the form Some A are B
Some B are C
∴ Some A are C

which is an invalid syllogism, even though its premises and conclusion are all true.

A syllogism of a given form is invalid if it is possible to produce another syllogism of the same form with true premises and a false conclusion. One can prove a syllogism invalid by generating counter-examples in this way.

Syllogistic arguments play a very important role in reasoning but are usually not stated explicitly in the forms discussed above.

2. Generalization of Lewis Carroll Diagrams

It is possible to extend Lewis Carroll diagrams to relations among four or more classes, so that we might treat extended syllogisms such as:

Some people are babies.
All babies are young animals.
All young animals are delicate.
∴Some people are delicate.

However, when the diagrams are thus extended, they lose their transparency. The feature which makes them especially useful for the three classes of ordinary syllogisms is the fact that an O "sitting on the fence" actually overlaps contiguous areas. This is not always possible for diagrams representing a larger number of classes.

For a generalized syllogism, which may contain a large number of classes and premises, we will replace the Lewis Carroll diagram by a linear array, simply a list of all possible combinations of classes and class complements. For example, the list for two classes, A B, is:

A B
A not-B
not-A B
not-A not-B

The list corresponding to three classes has eight entries and the list for four classes has sixteen. The markers which were used in the Lewis Carroll diagrams are now placed next to the corresponding entries in the list. Thus, we represent the statement "no A are B" by placing an X next to each entry in the list containing A and B unnegated.

e.g., A B C X
A B not-C X

The statement "so e A are B" was previously represented by a circle on the fence between the regions A, B, C and A, B, not-C. Now the regions have been replaced by entries on a list with no topological significance. Further, there may be more than three classes, so that the statement above might refer to several entries, e.g., A B C D, A B C not-D, A B not-C D, and A B not-C not-D. We need a new nongraphical notation to represent this situation. We can put a marker, say 1, next to each entry representing a class with A, B unnegated. In the above example, each of the four entries given would receive a "1". If another statement, say "some B are not-C", put another O "on the fence", we might use a "2" to indicate each class intersected by this new O.

The increase in number of classes also forces some operational changes. In the Lewis Carroll diagrams an O on the fence always intersected exactly two regions. If a later statement put an X in one of the resions, the O was automatically taken off the fence and placed in the other region, since there was no other possibility. This is no longer the case when a "fence" joins more than two regions. In the above example, if an X were placed next to A B C D (say by the statement "no C are D"), the 1 next to this entry would be eliminated, but three 1's would still remain.

Thus, the 1 is still on the fence. It is not replaced by an 0 unless a single 1 remains on the list.

3. Generating Logic "Diagrams"

We are ready to translate the processes involved in the use of generalized Lewis Carroll diagrams into the form of computer programs. There are four major parts to this undertaking. First, in this section, we will discuss the creation of generalized diagrams, given a list of objects. Then we can write procedures which "mark" diagrams to represent the statements given as their input. Once this is done, and a diagram representing the set of all premisses is made, we can then test the validity of a further statement with respect to these premisses. These three preceding steps, in fact, create a complete, consistent structure. Such a structure, however, will only accept statements and objects in a rigidly standardized form. Our fourth and last step will be, therefore, to write a set of procedures which enable us to deal with a more "relaxed" set of statements and objects.

We have seen how the Lewis Carroll diagrams can be generalized, by simply making a list of the names of the regions, without placing any topological interpretations on such a list. The "contents" of each region can then be denoted simply by associating these with the appropriate name. This is done most simply by making the contents the THING of the region name. The list of region names is best written as a LOGO sentence with commas (or some

other demarcation) between the names. In that way, we can still use sentences for region names. This convention does not permit the individual objects within a region name (NOT GREEN CATS, for example) to be sentences, but, rather than use another symbol as a further level of demarcation, we will make these objects into LOGO words by putting dashes in place of the spaces within them. (NOT-GREEN-CATS, for example). Thus, if we are given a set of objects we first insert dashes in each:

```
TO DASH /OBJECT/
1Ø TEST WORDP /OBJECT/
2Ø IF TRUE OUTPUT /OBJECT/
3Ø OUTPUT DASHSENTENCE /OBJECT/
END

TO DASHSENTENCE /SENTENCE/
1Ø TEST IS COUNT /SENTENCE/ 1
2Ø IF TRUE OUTPUT FIRST /SENTENCE/     (If there is just one word
                                        in /SENTENCE/, we get this
                                        remaining word using FIRST)
3Ø OUTPUT WORD WORD
     FIRST /SENTENCE/
     "_"
     DASHSENTENCE BUTFIRST /SENTENCE/
END

←PRINT DASH "IS A MAN"
IS-A-MAN
←PRINT DASH "ARISTOTLE"
ARISTOTLE
←
```

The really important task, at this early point, is construction of the region name list. We will have a list of objects, each of which is a LOGO word (we ensure this by using DASH). The procedure REGIONLIST takes /OBJECTS/ as input and outputs the list of all possible combinations, each containing every object

or its negation (NOT-object). These possibilities are to be separated by commas. Let us consider an example where the objects are taken as A, B, C and we place each possibility on a separate line.

A	B	C
A	B	NOT-C
A	NOT-B	C
A	NOT-B	NOT-C
NOT-A	B	C
NOT-A	B	NOT-C
NOT-A	NOT-B	C
NOT-A	NOT-B	NOT-C

There are a number of ways to systematically generate this set of combinations. Furthermore, the result of each method can be looked at in different ways, depending on the patterns one sees in the result. The scheme used in the example above was to find all possibilities, changing the rightmost character (lines 1, 2), to repeat these possibilities, having changed the next character (lines 3 and 4), and so on. This process is very reminiscent of counting, if one thinks of an automobile odometer, for example, and there is, in fact, an equivalence between the scheme used and counting in binary mode. Counting to base 2 is governed by the rules $0 + 0 = 0$ $0 + 1 = 1 + 0 = 1$ and $1 + 1 = 10$ where "carrying" is exactly as in base ten. Thus, putting zeros in front of the binary numbers (to give 3 digits) as we count from 0 to 7,

000
001
010
011
100
101
110
111

The zeros and ones exactly reproduce the pattern of "NOT"s and absence of NOTs if we make the correspondence 0⟷absence of NOT 1⟷NOT.

Thus, counting in binary (which is very easy to program) gives us the pattern we need for our combination generating problem. To get the region name list for /N/ objects we need only count from $\underbrace{00\ldots0}_{/N/}$ to $\underbrace{11111\ldots1}_{N}$ in binary, then use each number to create a region name, using the correspondence given above. Thus, we write a procedure, which, given a binary number, outputs the next binary number.

```
TO BINARYNEXT /NUMBER/
1Ø TEST EMPTYP /NUMBER/
2Ø IF TRUE OUTPUT 1
3Ø TEST ZEROP LAST /NUMBER/      (If the last digit is zero, just
4Ø IF TRUE OUTPUT WORD            change it to 1)
          BUTLAST /NUMBER/
          1
5Ø OUTPUT WORD                                (If the last digit is 1 (the
          BINARYNEXT BUTLAST /NUMBER/ only other possibility) change
          Ø                                    it to zero and add 1 to the
                                               binary number, BUTLAST
END                                            /NUMBER/.)
```

This process continues until either a zero is found in which case it is replaced by a 1, or until the last digit of the original number has been passed, in which case a 1 is placed in front.

```
←PRINT BINARYNEXT ØØ
Ø1
←PRINT BINARYNEXT Ø1
1Ø
←PRINT BINARYNEXT 11
1ØØ
←
```

We want to start the process with /N/ zeros so,

```
TO ZERO /N/
1Ø TEST IS /N/ Ø
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT WORD OF
          Ø
          ZERO (DIFF /N/ AND 1)
END

←PRINT ZERO "3"
ØØØ
←PRINT BINARYNEXT BINARYNEXT ZERO 3
Ø1Ø
```

We use these two procedures in two further procedures, one of which, REGIONLIST, starts things off, and the other, ADDTOLIST, adds one combination at a time to the list of region names.

```
TO ADDTOLIST /OBJECTS/ /BINARY #/
1Ø TEST GREATERP (COUNT /BINARY #/)
     (COUNT /OBJECTS/)          (If the number of digits in the
2Ø IF TRUE OUTPUT /EMPTY/        binary number exceeds the number
                                 of objects, we are finished)
3Ø OUTPUT SENTENCE SENTENCE
     NOTT /BINARY #/ /OBJECTS/ (NOTT gives the combination of
     ","                        /OBJECTS/ corresponding to /BINARY #/)
     ADDTOLIST /OBJECTS/
               (BINARYNEXT /BINARY#/)
END
```

The procedure NOTT /NUMBER/ /LIST/, which applies the correspondence between NOTs and a binary number to /LIST/ is straightforward: If the first digit of /NUMBER/ is Ø, we leave the first element of list unchanged. Otherwise, if it is 1, we put a "NOT-" in front of it. We repeat the process until the list is exhausted.

```
TO NOTT /NUMBER/ /LIST/
1Ø TEST IS /LIST/ /EMPTY/          (If the list is /EMPTY/, we are
2Ø IF TRUE OUTPUT /EMPTY/           finished)
3Ø TEST IS FIRST /NUMBER/ Ø
4Ø IF TRUE OUTPUT SENTENCE
        FIRST /LIST/
        NOTT (BUTFIRST /NUMBER/) (BUTFIRST /LIST/)
5Ø OUTPUT SENTENCE
        WORD "NOT-" (FIRST /LIST/)
        NOTT (BUTFIRST /NUMBER/) (BUTFIRST /LIST/)
END


←PRINT NOTT "Ø11" "DOGS CATS ELEPHANTS"
DOGS NOT-CATS NOT-ELEPHANTS
```

Now, REGIONLIST is easy:

```
TO REGIONLIST /OBJECTS/
1Ø OUTPUT ADDTOLIST /OBJECTS/ (ZERO OF COUNT OF /OBJECTS/)
END

←PRINT REGIONLIST "A B C"
A B C , A B NOT-C , A NOT-B C , A NOT-B NOT-C , NOT-A B C , NOT-A B
NOT-C , NOT-A NOT-B C , NOT-A NOT-B NOT-C ,

←PRINT REGIONLIST "CATS DOGS MONKEYS ELEPHANTS CAMELS"
CATS DOGS MONKEYS ELEPHANTS CAMELS , CATS DOGS MONKEYS ELEPHANTS
NOT-CAMELS , CATS DOGS MONKEYS NOT-ELEPHANTS CAMELS , CATS DOGS MONKEYS
NOT-ELEPHANTS NOT-CAMELS , CATS DOGS NOT-MONKEYS ELEPHANTS CAMELS , CATS
DOGS NOT-MONKEYS ELEPHANTS NOT-CAMELS , CATS DOGS NOT-MONKEYS
NOT-ELEPHANTS CAMELS , CATS DOGS NOT-MONKEYS NOT-ELEPHANTS NOT-CAMELS ,
CATS NOT-DOGS MONKEYS ELEPHANTS CAMELS , CATS NOT-DOGS MONKEYS ELEPHANTS
NOT-CAMELS , CATS NOT-DOGS MONKEYS NOT-ELEPHANTS CAMELS , CATS NOT-DOGS
MONKEYS NOT-ELEPHANTS NOT-CAMELS , CATS NOT-DOGS NOT-MONKEYS ELEPHANTS
CAMELS , CATS NOT-DOGS NOT-MONKEYS ELEPHANTS NOT-CAMELS , CATS NOT-DOGS
NOT-MONKEYS NOT-ELEPHANTS CAMELS , CATS NOT-DOGSNOT-MONKEYS
NOT-ELEPHANTS NOT-CAMELS , NOT-CATS DOGS MONKEYS ELEPHANTS CAMELS ,
NOT-CATS DOGS MONKEYS ELEPHANTS NOT-CAMELS , NOT-CATS DOGS MONKEYS
NOT-ELEPHANTS CAMELS , NOT-CATS DOGS MONKEYS NOT-ELEPHANTS NOT-CAMELS ,
NOT-CATS DOGS NOT-MONKEYS ELEPHANTS CAMELS , NOT-CATS DOGS NOT-MONKEYS
ELEPHANTS NOT-CAMELS , NOT-CATS DOGS NOT-MONKEYS NOT-ELEPHANTS CAMELS ,
NOT-CATS DOGS NOT-MONKEYS NOT-ELEPHANTS NOT-CAMELS , NOT-CATS NOT-DOGS
MONKEYS ELEPHANTS CAMELS , NOT-CATS NOT-DOGS MONKEYS ELEPHANTS
NOT-CAMELS , NOT-CATS NOT-DOGS MONKEYS NOT-ELEPHANTS CAMELS , NOT-CATS
NOT-DOGS MONKEYS NOT-ELEPHANTS NOT-CAMELS , NOT-CATS NOT-DOGS
NOT-MONKEYS ELEPHANTS CAMELS , NOT-CATS NOT-DOGS NOT-MONKEYS ELEPHANTS
NOT-CAMELS , NOT-CATS NOT-DOGS NOT-MONKEYS NOT-ELEPHANTS CAMELS ,
NOT-CATS NOT-DOGS NOT-MONKEYS NOT-ELEPHANTS NOT-CAMELS ,
←
```

This completes the first stage of our program--we can easily create region name lists. These lists are not easy to work with directly in LOGO since the separators of interest are commas, and not spaces. FIRST, for example, will give us the first element of the first region name. We therefore write a few simple procedures which will be useful in working with our new kind of list.

```
TO PULL /LIST/                          (gives everything to first comma)
1Ø TEST EITHER
             EMPTYP /LIST/
             IS FIRST /LIST/ ","
2Ø IF TRUE OUTPUT /EMPTY/
3Ø OUTPUT SENTENCE
             FIRST /LIST/
             PULL BUTFIRST /LIST/
END


TO REST /LIST/                          (gives everything after first comma)
1Ø TEST EITHER
             EMPTYP /LIST/
             IS FIRST /LIST/ ","
2Ø IF TRUE OUTPUT BUTFIRST /LIST/
3Ø OUTPUT REST (BUTFIRST /LIST/)
END


TO NUMBER /LIST/                        (counts "region names" by counting
1Ø TEST EMPTYP /LIST/                    commas -- this is correct since
2Ø IF TRUE OUTPUT Ø                      the list ends with a comma.)
3Ø TEST IS (FIRST /LIST/) ","
4Ø IF TRUE OUTPUT SUM
             1
             NUMBER (BUTFIRST /LIST/)
5Ø OUTPUT NUMBER (BUTFIRST /LIST/)
END
```

These procedures enable us, for example, to write a simple procedure NICEPRINT to print the list of regions "nicely" by putting each on a separate row. To make NICEPRINT even more useful, next to each

region we print its "contents", given by the THING of the region name. We have not yet "marked" any of the regions, so at the moment, the contents are all /EMPTY/.

```
TO NICEPRINT /LIST/
1Ø TEST EMPTYP REST /LIST/
2Ø IF TRUE STOP
3Ø TYPE PULL /LIST/                    (TYPE prints but does not go to
4Ø TYPE ","                             the beginning of the next line)
5Ø PRINT THING (PULL /LIST/)
6Ø NICEPRINT (REST /LIST/)
END
```

```
←NICEPRINT REGIONLIST "FRED HARRY JANE SALLY"
FRED HARRY JANE SALLY,
FRED HARRY JANE NOT-SALLY,
FRED HARRY NOT-JANE SALLY,
FRED HARRY NOT-JANE NOT-SALLY,
FRED NOT-HARRY JANE SALLY,
FRED NOT-HARRY JANE NOT-SALLY,
FRED NOT-HARRY NOT-JANE SALLY,
FRED NOT-HARRY NOT-JANE NOT-SALLY,
NOT-FRED HARRY JANE SALLY,
NOT-FRED HARRY JANE NOT-SALLY,
NOT-FRED HARRY NOT-JANE SALLY,
NOT-FRED HARRY NOT-JANE NOT-SALLY,
NOT-FRED NOT-HARRY JANE SALLY,
NOT-FRED NOT-HARRY JANE NOT-SALLY,
NOT-FRED NOT-HARRY NOT-JANE SALLY,
NOT-FRED NOT-HARRY NOT-JANE NOT-SALLY,
←
```

4. Marking the "Diagrams"

There are two different operations required to represent a statement, or sequence of statements on a diagram, whether it is of the Lewis Carroll type or a more general list one. We must be able to "mark off" regions, by placing X's in them. We do this by making the THING of the region name "X". We also must be able to denote occupancy or possible occupancy of a region. If there is only one available region satisfying the given constraints in the statement(s), an "0" is placed within it. If several regions are possible, they are "joined" by placing the same number in the THING of each. Any conflict between 0s and Xs means that a contradiction has been found.

Let us first consider the algorithm required for "joining" or, in terms of Lewis Carroll diagrams, placing an "0" on the fence. We take each region name and determine whether it contains the combination of objects and their negation as in the statement we are joining. For example, given the three objects A, B, C and the list of 8 names that they generate, we determine that the elements A NOT-B are contained within the two regions A NOT-B C, A NOT-B NOT-C. We make a list of all such "valid" names, which do not contain an "X". If there is exactly one such region, we *occupy* it by adding an "0" to its THING. If there is more than one region possible, we *include* the current join number within the THING of each. Of course, if no regions are possible, a contradictory set of premisses is involved and we can go no further.

Our first step is to write a procedure SUBSETP, which, given two sets /A/ and /B/ as inputs, outputs TRUE or FALSE as all elements of /A/ are or are not contained among those of /B/. SUBSETP in

turn depends on CP /EL/ /LIST/ which determines whether a given single element /EL/ is contained in a given list, /LIST/.

```
TO SUBSETP /A/ /B/
1Ø TEST EMPTYP /A/
2Ø IF TRUE OUTPUT "TRUE"          (is the first element of /A/ contained
3Ø TEST CP (FIRST /A/) /B/        in /B/?)
4Ø IF FALSE OUTPUT "FALSE"        (if not /A/ is not a subset of /B/)
5Ø OUTPUT SUBSETP (BUTFIRST /A/) /B/ (if yes, continue the process
                                      with the rest of /A/)
END


TO CP /ELEMENT/ /LIST/
1Ø TEST EMPTYP /LIST/             (have we gone through the whole list?)
2Ø IF TRUE OUTPUT "FALSE"         (if so /ELEMENT/ is not contained)
3Ø TEST IS /ELEMENT/
           FIRST /LIST/
4Ø IF TRUE OUTPUT "TRUE"
5Ø OUTPUT CP /ELEMENT/ (BUTFIRST /LIST/)
END


←PRINT SUBSETP "A B C D" "D C B A"
TRUE
←PRINT SUBSETP "PLEAT"  "ELEPHANT"
TRUE
```

We next write the main procedure JOIN /ELEMENTS/ /JOIN #/ /LIST/, and its sub-procedures. JOIN is virtually a line by line translation of the joining algorithm just stated.

```
TO JOIN /ELEMENTS/ /JOIN #/ /LIST/
1Ø MAKE "COMMON REGIONS" COMMON OF /ELEMENTS/ /LIST/
         (make a list of regions on /LIST/ without "X"s, having
         /ELEMENTS/ as subset)
2Ø TEST IS NUMBER /COMMON REGIONS/ Ø
3Ø IF TRUE EXIT "CONTRADICTION" (EXIT is a LOGO built in which
                                 halts execution and causes its
                                 input to be printed)
4Ø TEST IS NUMBER /COMMON REGIONS/ 1
5Ø IF TRUE OCCUPY /JOIN #/  (PULL /LIST/)
                                 (if there is exactly one region,
                                 place an "0" within it)
6Ø IF TRUE STOP
7Ø INCLUDE /JOIN #/ /COMMON REGIONS/ (otherwise place /JOIN #/ within
                                      each of the common regions.)
END
```

The procedures remaining to be written are COMMON, OCCUPY and INCLUDE, all mentioned in our description of the joining algorithm.

```
TO COMMON /ELEMENTS/ /LIST/
1Ø TEST EMPTYP /LIST/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST BOTH
       SUBSETP /ELEMENTS/ (PULL /LIST/)
       NOT CP "X" (THING OF PULL /LIST/)
4Ø IF TRUE OUTPUT SENTENCE SENTENCE
       PULL /LIST/
       ","
       COMMON /ELEMENTS/ (REST /LIST/)

5Ø OUTPUT COMMON /EL/ (REST /LIST/)
END
```

(PULL /LIST/ is the first region name on /LIST/. If /ELEMENTS/ is a subset of the first region name and the region does not contain "X", add the first region to the output list) (Otherwise, just repeat rest of /LIST/)

NOT, on line 3Ø simply outputs the negation of its input,

```
TO NOT /INPUT/
1Ø TEST IS INPUT "TRUE"
2Ø IF TRUE OUTPUT "FALSE"
3Ø OUTPUT "TRUE"
END
```

The following example illustrates the use of COMMON.

```
←MAKE "REGIONS" REGIONLIST "CATS HATS BATS"
←NICEPRINT /REGIONS/
CATS HATS BATS,
CATS HATS NOT-BATS,
CATS NOT-HATS BATS,
CATS NOT-HATS NOT-BATS,
NOT-CATS HATS BATS,
NOT-CATS HATS NOT-BATS,
NOT-CATS NOT-HATS BATS,
NOT-CATS NOT-HATS NOT-BATS,
←PRINT COMMON "CATS HATS" /REGIONS/
CATS HATS BATS , CATS HATS NOT-BATS ,
←PRINT COMMON "NOT-BATS" /REGIONS/
CATS HATS NOT-BATS , CATS NOT-HATS NOT-BATS , NOT-CATS HATS NOT-BATS ,
NOT-CATS NOT-HATS NOT-BATS ,
←
```

We look ahead and write a version of OCCUPY which is just a little more powerful than we require at the moment. Right now, we need only have OCCUPY place an 0 in the THING of the region name given as input. Later, however, we will also want to change a "join" to an "0" when marking "X"s reduces the number of possible regions to 1. In this latter case we want to DELETE an inputted "join number" as well as write in an "0".

```
TO OCCUPY /JOIN #/ /REGION/
1Ø MAKE /REGION/
        SENTENCE
           "0"
           DELETE /JOIN #/ (THING OF /REGION/)
END
```

DELETE goes through /LIST/, deleting each occurrence of /ELEMENT/.

```
TO DELETE /ELEMENT/ /LIST/
1Ø TEST EMPTYP /LIST/
2Ø IF TRUE OUTPUT /EMPTY/
3Ø TEST IS /ELEMENT/ (FIRST /LIST/)
4Ø IF TRUE OUTPUT DELETE (BUTFIRST /LIST/)
5Ø OUTPUT SENTENCE
             FIRST /LIST/
             DELETE (BUTFIRST /LIST/)
END


←DELETE "B" "A B C B"
A C
```

To complete the joining algorithm we need only write INCLUDE /NUMBER/ /LIST/. INCLUDE simply includes /NUMBER/ as a member of the THING of each region name on /LIST/.

```
TO INCLUDE /NUMBER/ /LIST/
1Ø TEST EMPTYP /LIST/
2Ø IF TRUE STOP
3Ø MAKE PULL /LIST/
        SENTENCE
            THING OF PULL /LIST/
            /NUMBER/
4Ø INCLUDE /NUMBER/ (REST /LIST/)
END
```

And we have completed our implementation of the joining algorithm. We try it out; using the list /REGIONS/ we generated just before:

```
←JOIN "CATS HATS" 1 /REGIONS/
←JOIN "CATS NOT-BATS" 2 /REGIONS/
←NICEPRINT /REGIONS/
CATS HATS BATS,1
CATS HATS NOT-BATS,1 2
CATS NOT-HATS BATS,
CATS NOT-HATS NOT-BATS,2
NOT-CATS HATS BATS,
NOT-CATS HATS NOT-BATS,
NOT-CATS NOT-HATS BATS,
NOT-CATS NOT-HATS NOT-BATS,
←
```

The other operation we will have to perform on our extended syllogism diagrams is that of "marking off" regions to indicate the impossibility of their being occupied. As in the case of joining, we will generally have several regions to be marked off, as the result of a given premiss, each containing the elements specified in the premiss. If a region does not contain anything, we need merely put an "X" in it. If the region contains "0", we have a contradiction. The only difficult case is when the region contains one or more join numbers. When this is true, we use a procedure UNJOIN to "remove" all joins from this region. To remove each join number, /JOIN #/, we make a list, using JOINLIST, of all other regions containing /JOIN #/. If there is exactly one region on this list, /JOIN #/ there is replaced by "0". If

there is more than one region on the list, these other regions are not affected. In either case /JOIN #/ can then be removed from the given region. When all join numbers have been removed from the given region, an "X" can be inserted.

For example, using the small region list corresponding to "A B", marking off all regions containing the element "A" in

```
A       B       ,   1
A       NOT-B   ,   1
NOT-A   B       ,   1
NOT-A   NOT-B   ,   1
```

results in

```
A       B       ,   X
A       NOT-B   ,   X
NOT-A   B       ,   1
NOT-A   NOT-B   ,   1
```

and, doing the same in

```
A       B       ,   1
A       NOT-B   ,   1
NOT-A   B       ,   1
NOT-A   NOT-B   ,
```

results in

```
A       B       ,   X
A       NOT-B   ,   X
NOT-A   B       ,   0
NOT-A   NOT-B   ,
```